UNIVERSITAT POLITECNICA DE CATALUNYA

PARALLELISM

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

15th May 2019, Q2

Contents

1 Introduction

2 Anlysis with *Tareador*

In order to study which is the best approach that we should use to parallelize the code (using the *OpenMP* directives), a study about the task dependencies is needed.

In the following subsections, we will analyse the functions *merge* and *multisort* of the given code using *Tareador* to see the potential parallelism they have and also their dependences. As both functions are recursive, it is very likely that we will be able to parallelise them.

The modified code can be found in the *multisort-tareador.c* file, inside the codes directory.

2.1 Merge Function

The merge function is responsible for sorting two vectors, merging them into one. This is done using recursive calls that divide the vectors into smaller ones.

Our study in this function focuses on the two recursive calls done in the else fragment of the function. A task is created every time we make a recursive call to the function. As a consequence, some created tasks will also execute the basicmerge function.

To do that, we have used the *tareador_start_task* and *tareador_end_task* functions. The resulting code is shown below.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
        long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        tareador_start_task("me1");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("me1");

        tareador_start_task("me2");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("me2");
    }
}
```

Figure 1: Modified version of the merge function.

2.2 Multisort Function

The multisort function divides the input vector into four smaller vectors, apply a recursive call for each of them and them merges the 4 vectors into a big sorted one.

We will use the same strategy than in the merge function. This is, creating a task for each recursive call and for each call to the merge function.

We have also used the *tareador_start_task* and *tareador_end_task* functions. The code is the following:

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("mu1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("mu1");

        tareador_start_task("mu2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("mu2");

        tareador_start_task("mu3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("mu3");

        tareador_start_task("mu4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("mu4");

        tareador_start_task("me1");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("me1");

        tareador_start_task("me2");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("me2");


        tareador_start_task("me3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("me3");

    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 2: Modified version of the multisort function.

2.3 Task dependency graph (TDG)

We can clearly see that Figure **??** is divided into two parts: the multisort and the merge.

As we said before, in the multisort function we divide the input vector into 4 smaller vectors, until the vector size is smaller than $MIN\_SORT\_SIZE \times 4L$. This can be seen at the top of the figure, as we can see 4 different boxes (green, red, yellow and purple). Moreover, there is no data sharing between them, so we can execute them at the same time.

Then, there is a new recursion level inside each box. These four calls reach the limit value, so they do a *basicmerge* call. Afterwards, we need the first two childs to terminate before executing the first merge call (and the same for the third and fourth childs with the second merge call). Thus, there exist a dependence between the recursive calls to multisort and the calls to the first two merge functions. Finally, the third merge call can be executed only when the previous two merge calls have terminated (another dependence).

On the other hand, in the merge function, we create tasks when the number of elements of the vector that we want to sort is bigger or equal than $MIN\_MERGE\_SIZE \times 2L$. We see in the figure that the two created tasks do not have any dependence between them. A *basicmerge* call will be done when we reach the limit value.

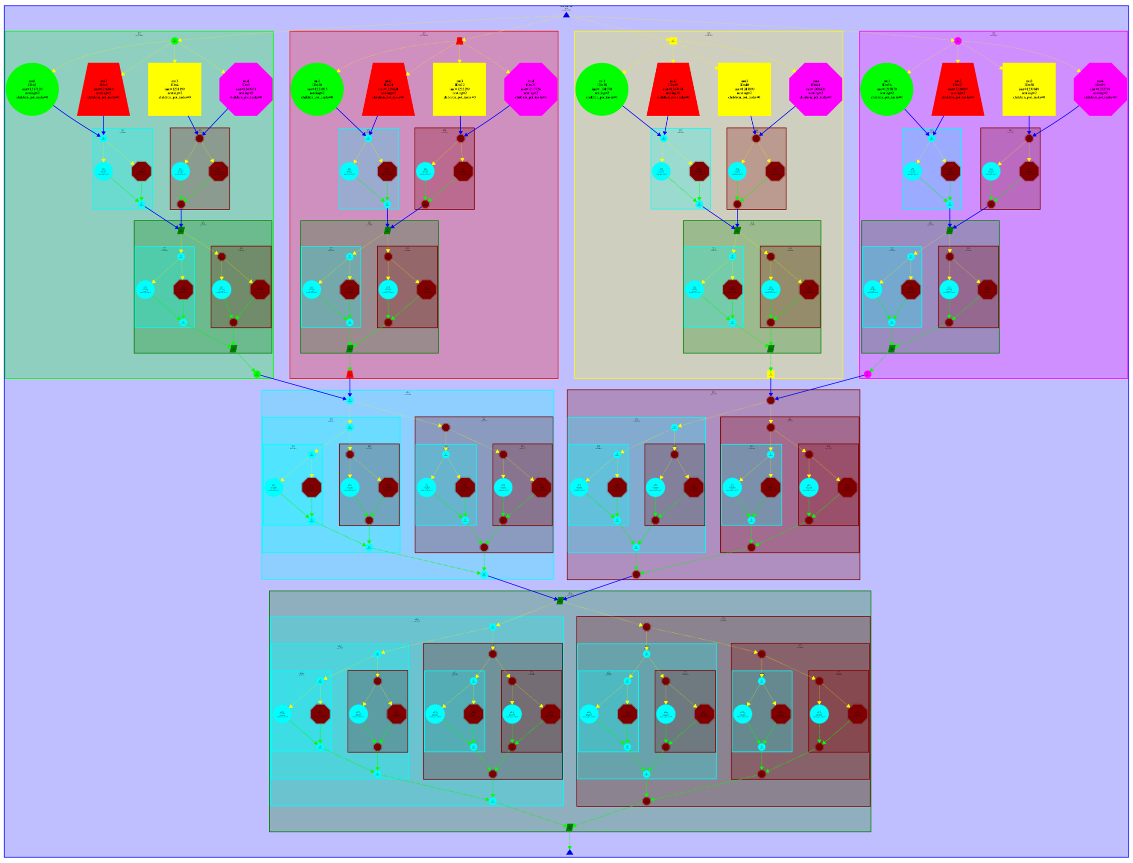

Figure 3: Task dependency graph obtained using *Tareador*.

With all these information, we can conclude that we can use a **Tree strategy** (generate a task for each recursive function) because there are no dependences between sibling tasks. Besides, we can also use a **Leaf strategy** (generate a task in each base case of the functions).

Furthermore, we must be careful about the synchronization we need between the multisort tasks and the merge ones because, as we said before, the first merge call depends on the first two multisort calls, the second merge call depends on the thids and fourth multisort calls and the third merge call depends on the previous two merge calls. Without synchronization we would have wrong results due to data racing.

2.4 Parallel Performance and Scalability

The following table shows the execution time and speed-up values obtained when varying the number of processors.

Processors	Execution Time [ns]	Speed-Up
1	20334411001	1
2	10173716001	1.99872013323365
4	5086725001	3.99754478510288
8	2550595001	7.97241858979085
16	1289922001	15.7640624667507
32	1289909001	15.7642213406029
64	1289909001	15.7642213406029

Table 1: Execution Time and Speed-Up varying the number of processors.

We can see that until 16 processors, the results obtained are pretty close to the ideal ones. However, from 16 to infinite processors there is no improvement on the speed-up values. This is because in Figure **??** we see that the maximum number of tasks executed at the same time are 16. For this reason, using more than 16 processors will not have any impact on the execution time and speed-up.

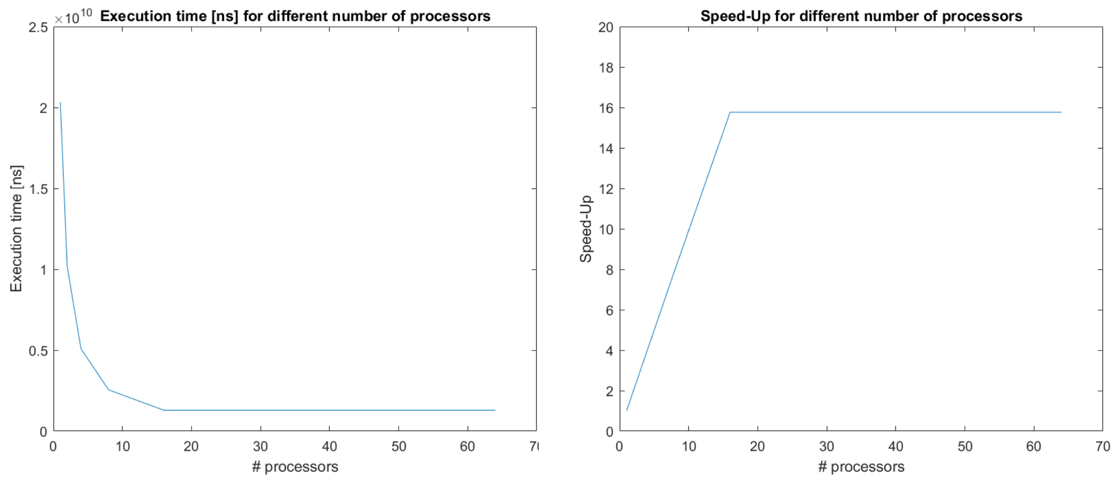

Figure 4: Execution Time and Speed-Up plots varying the number of processors.

Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

3 Annex

3.1 Scalability anaysis: Traces using different number of processors

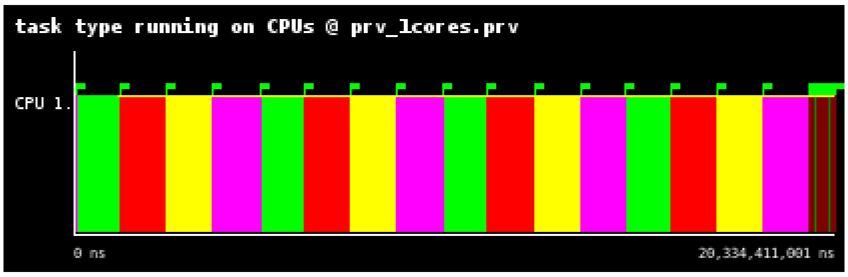

Figure 5: Execution flow of the code using 1 processor.

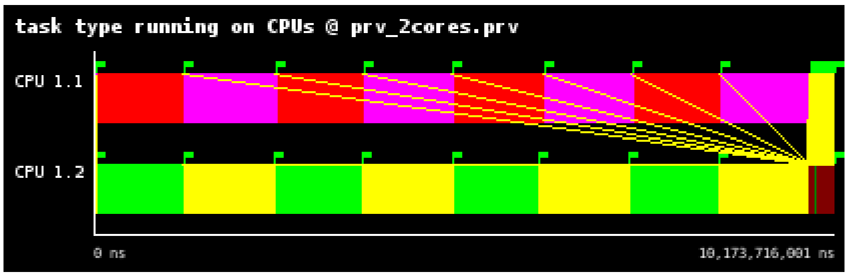

Figure 6: Execution flow of the code using 2 processors.

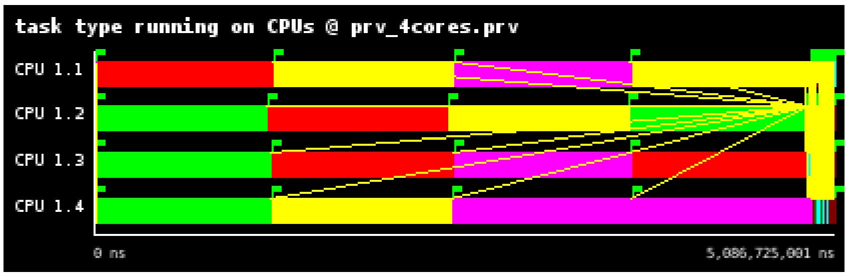

Figure 7: Execution flow of the code using 4 processors.

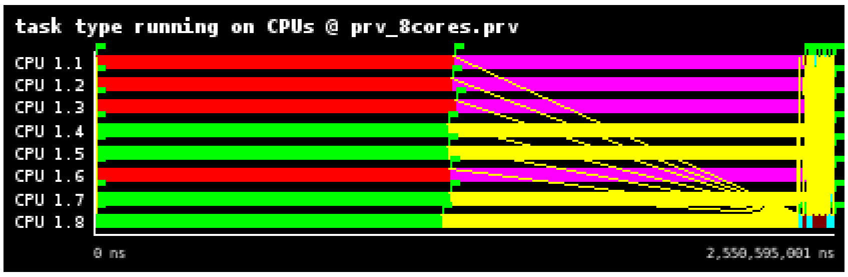

Figure 8: Execution flow of the code using 8 processors.

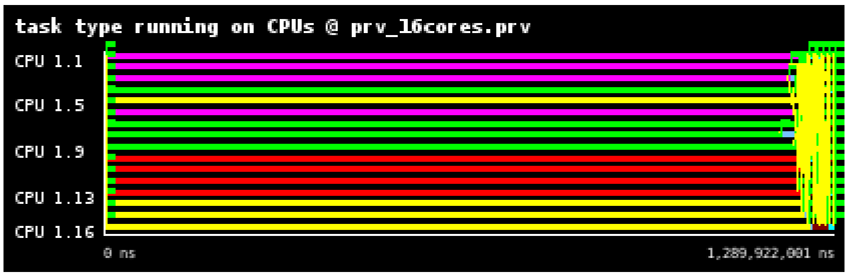

Figure 9: Execution flow of the code using 16 processors.

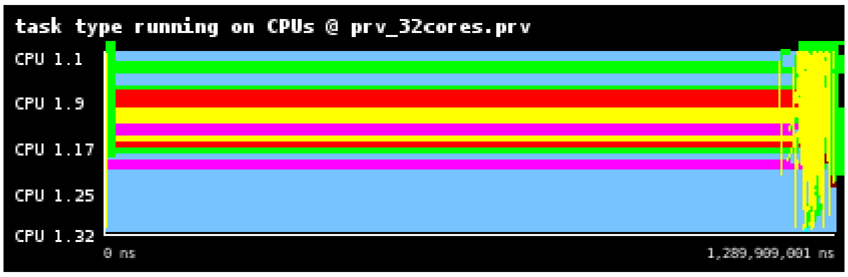

Figure 10: Execution flow of the code using 32 processors.

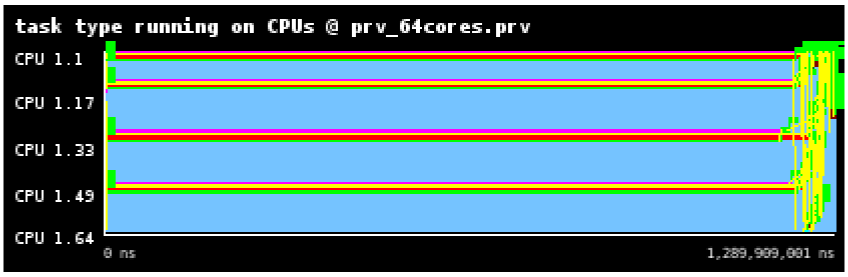

Figure 11: Execution flow of the code using 64 processors.